Applying Formal Methods and Object-Oriented Analysis to Existing Space Shuttle Software*

Betty H. C. Cheng†
Michigan State University
Department of Computer Science
East Lansing, MI 48824-1027
chengb@cps.msu.edu

Brent Auernheimer‡
California State University, Fresno
Department of Computer Science
Fresno, CA 93740-0109
brent_auernheimer@CSUFresno.edu

Abstract

Correctness is the most important issue in safety-critical software control systems. Unfortunately, failures in critical segments of software for medical radiation treatment, communications, and defense are familiar to the public. Such incidents motivate the use of software development techniques that reduce errors and detect defects. The benefits of applying formal methods in requirements-driven software development (forward engineering) are well-documented; formal notations are precise, verifiable, and facilitate automated processing. This paper describes the application of formal methods and object-oriented modeling to *reverse* engineering, in which formal specifications are developed for existing, or *legacy*, code. In this project, several layers of formal specifications were constructed for a portion of the NASA Space Shuttle Digital Auto Pilot (DAP), a software module that is used to control the position of the spacecraft through appropriate jet firings. The reverse engineering process was facilitated by the *Object* Modeling *Technique* (OMT), an informal software development approach that uses graphical notations to describe software requirements.

1 Introduction

Correctness is most important and necessary in safety-critical software control systems [1], Critical software failures in medical radiation equipment [2], communication networks, and defense systems are familiar to the public. The large number of software malfunctions regularly reported to the software engineering community [3], new statutes concerning liability for such failures, and a recent National Research Council Aeronautics and Space Engineering Board Report [4], additionally motivate the use of software development techniques that reduce errors and detect defects.

*The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the National Aeronautics and Space Administration. Additionally, the authors' work on this project was supported by NASA/ASEE Summer Faculty fellowships. A preliminary version of this paper was presented at the NASA/Goddard Software Engineering Workshop, Greenbelt, Maryland, December, 1993.

†This author is also supported in part by NSF grant CCR-9209873.

‡This author gratefully acknowledges the Software Engineering Institute at Carnegie Mellon University for support as a Visiting Scientist, Spring 1994.

The benefits of using formal methods in requirements-d riven software development (forward engineering) are well-documcmted [5, 6]. A formal method is characterized by a formal specification language and a set of rules governing the manipulation of expressions in that language. Traditionally, formal methods have been used in the early phases of development, in order to describe the requirements of a soft ware system or component. Using formal specification languages facilitates the carl y evaluation of a software design and verification of its implementation through the use of formal reasoning techniques [7, 8]. A formal specification can be manipulated, using automated techniques, to enable the designer to assess the consistency, completeness, and robustness of a design before it is implemented. Each step in the development process can be justified by mathematical proof, thus minimizing the number of errors due to misinterpretation and ambiguity.

Re-engineering is the process of examining, understanding, and modifying a system with the intent of implementing the system in a new form [9]. Re-engineering of existing, or *legacy*, code is preferred to redeveloping the software from the original requirements in order to preserve functionalist y that has been achieved over a period of time and to provide continuity to current users of the software [1 O]. Onc of the most difficult aspects of re-engineering is the recognition of the function of the existing programs. *Reverse Engineering* is the process of constructing high level representations from lower level instantiation of an existing systcm. Common reverse engineering methods used by software m aintenan ce engineers are observation (for example, test case analysis) and examination of source code. These techniques are often tedious and error-prone.

Onc way to take advantage of the benefits of formal methods in legacy systems, is to reverse engineer the existing program code into formal specifications [11,12, *1 3]*. The resulting formal specificat ions can then be used as the basis for change requests and the foundation for subsequent verification and validation [1-4]. Considering the high cost of re-implementation and the need to preserve critical functionality, reverse engineering of code into formal specific.ations offers an alternative to traditional ad hoc approaches to maintaining safety-critical systems.

A highly visible example of a legacy system is the software for the NASA Space Shuttle, which was conceived in the early 1970s and has been operational for over ten years [4]. Onc component of the Shuttle software is the flight software, which provides guidance, navigation, and control for the Space Shuttle while it is in orbit. The navigation function determines where. the shuttle is, the guidance

function determines where it should go next, and the control function determines how to effect the next move. While the vehicle is in orbit, the Digital Auto Pilot (11A]') software determines attitude and translation] adjustments, based on astronaut selections. *Attitude* refers to the rotational position of the vehicle in terms of roll, pitch, and yaw, and *translation* refers to the $x, y,$ and z coordinates of the vehicle. Figure 1 gives a pictorial representation of translation and attitude as they relate to the position of the shuttle.

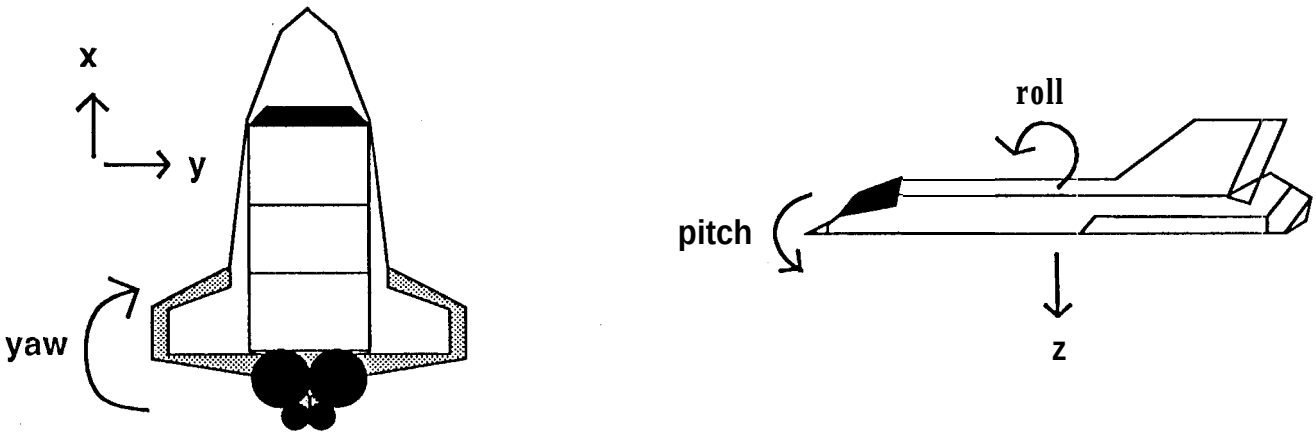

Figure 1: Shuttle Translational and Rotational Axes

Presently, the Space Shuttle flight software project has a well-ddincd process for managing requirements evaluation, This process is responsible for ensuring that requirements generated by an engineer are consistent, implementable, and will solve the problem at hand. However, this process does not in clu de a well-dcfind set of analytical methods and techniques [1 5, 14]. When a change is needed, a detailed description of the reasons for the change, known as a change *request* (CR), must be constructed before the system can be re-engineered to include the changes. Next, the requirements analyst performs an in-depth analysis of the CR, guided by a list of generic error categories, followed by a formal inspection of the CR by several representatives of the software project, including the author of the CR, requirements analyst, developer, verifier, and so on. Each potential error, termed an issue, that is identified by the requirements analyst or the inspection process remains '(open" until a clearly-described solution has been developed, at which point the issue is considered "closed ." When all inspect ions have been conducted for a CR and all issues have been closed, a CR is ready for implementation. At this point, a baseline for the project, a milestone that describes the current system with the accepted changes, is creat ed and scheduled for implement at ion.

The analysis step of the CR process involves studying, understanding, and analyzing the contents of a CR. Three major deficiencies in this process have been identified by requirements analysts [1 5]. First, there is no specific methodology for conducting the analysis of the CR. Scc.end, there are no specific completion criteria to indicate when sufficient information has been obtained for the CR. '1'bird, there is no specific structured mechanism for documenting the results of the analysis process. Moreover, since there is no structural approach for documenting the analysis, the understanding of the CR developed by the requirements analyst is not formally recorded for future use [1 5].

This paper describes a project that applies formal methods and ob jcc.t-oriented analysis to a subsystem of the DAP of the Shuttle, known as the *I'base Plane*, which determines whether jet firings arc needed to achieve tran slational or rot ational acceleration in a direction specified by the crew. More specificall y, the Phas e_Pl ane module was reverse engineered through the development of formal specifications that capture the details of Phase_Plane requirements. In order to facilitate the specification process, a pictorial description of the subsystem was constructed using the *Object* Modeling *Technique* (OMT) [16], an informal software development approach that uses graphical notations to describe software requirements,

The remainder of the paper is organized as follows. Section 2 gives a brief introduction to formal methods and object-oriental analysis techniques. Section 3 describes the Phase Plane project, including sample specifications and a discussion of the object-oriented analysis. Lessons learned from this project are described in Section 4, with a summary of the benefits of constructing formal specifications and t he use of ob ject-oriented analysis techniques in a reverse engineering project, Finally, conclusions and future investigations are described in Section 5.

2 Background Material

This section briefly defines and motivates the use of formal methods. Also, the benefits of object-oriented analysis is discussed.

2 . 1 Formal Methods

A *formal method* consists of a *formal specifica tion la nguage* and a set of *formally defined inference rules* [7]. The specification language is used to describe the intended system behavior, and the inference

rules provide a sound method for reasoning about the specifications. In general, formal methods in soft ware development provide man y benefits for forward engineering [5, 7]. First, it forces the designer to be thorough in the development and the documentation of a system design. Second, the developer is able to obtain precise answers to questions posed about the properties of the system. '1'bird, the developer is able to use automated reasoning to determine the correctness of the system (or a safety-critical component of the system) with respect to its specification.

Formal reasoning can be divided into two approaches: *program verification* and *program synthesis*. Program verification is the process of checking the semantics of program text against its specification. A program whose semantics satisfy its specification is said to be correct with respect to the specification. Program synthesis refers to formal techniques for systematically developing a program from a specification, such that the correctness of the resulting program (with respect to its specification) is inherent in the development process itself [17].

2.2 Object-Oriented Analysis

Software requirements define the objectives of a software development effort. They provide the basis by which the quality of the end-product is measured and guide the design of the software architecture. There are *a* variety of approaches to requirements analysis, many of them in the broad category known as *object-oriented requirements analysis* (OOA) [1 6]. An *object* is a self-contained module that includes both the data and procedures that act on that data. The emphasis on objects and their interactions in 00A is in contrast to the more traditional approach to software development, which focuses on proced u rcs,

Most 00A techniques begin by a careful assessment of the natural language description of the problem. A simple first step in developing an 00A model is to extract the *nouns* from the problem description. Many of these nouns will share common properties and may be described as instances of types, or classes. A *class* is *a* collection of objects that have common use. For example, *Galileo*, *Voyager*, and *Magellan* are all of the class spacecraft, and *Venus*, *Mars*, and *Mercury* are all of the class planet. Some classes, referred to as *subclasses* are specializations of other classes. For example, interplanetary spacecraft is a subclass of spacecraft.]n this manner, 00A is used to organize types into a class hierarchy based on a *isa* (as in '{an X is *a* Y") relationship.

It may be natural to think of an object as composed of other objects. For example, an interplanetary spacecraft may contain numerous jets, a guidance and navigation control system, and a probe to study a planet's atmosphere. This dependence introduces an additional dimension of the class hierarchy, namely, the *part of* relation. The parts of an object are often called its *attributes.*

As nouns can be used to identify candidate objects (and therefore, classes), verbs typically describe interactions between objects, therefore making them good candidates for operations, or *methods*, acting upon classes. Some verbs may describe a service for a particular class of ob jects, such as *fire* in the phrase "fire the jets." Other verbs may describe a possible state of an object, such as *coast* in the phrase "the spacecraft begins to coast." Therefore, verbs help to define the operations of a class as well as the dynamic behavior of the system as a whole.

in the early stages of software development, including object-oriented approaches, diagrams arc frequently used to describe requirements and guide development. The OMT []6] notation combines three complementary diagramming notations in order to document system requirements: object models, dynamic models, and functional models. The elements of a system that define its overall architecture are given by an object model, whose notation is similar to that used for cmtity-relationship diagrams used in database design. An object model determines the types of objects that can exist in the system and identifies allowable relationships among objects. As a result, the object model constrains the set of possible states that the system may enter. A *dynamic model* describes valid transitions between system states and indicates the conditions under which a state change may occur. Dynamic models are described in terms of state transition diagrams, A *functional model* is a data flow diagram that describes the computations to be performed by the system. Collectively, these three types of diagrams are used to model the properties of the system, including flow of control, flow of data, patterns of dependency, time sequence, and name-spa,cc relationships. The OMT approach is appealing since it offers multiple views of software requirements, and sin ce a single not at ion is not forced to describe many different perspectives of a given system, the notation for each type of diagram is simple to usc and easy to understand.

3 Project Description

Due to the criticality and the volume of flight system software, many recent flight system projects are incorporating formal methods into the software development process [1, 6]. in order to apply

formal methods to legacy flight software, however, reverse engineering is needed. The project described herein is associated with a larger multi-NASA site project to apply formal methods to a portion of the flight control software for the NASA Space Shuttle [14, 15]. The project described here uses formal methods and object-oricntwl analysis to reverse engineer the `Phase_Plane` module, which is the subsystem that provides automatic attitude control of the Shuttle. The criteria that led to the selection of `Phase_Plane` included finding a module whose requirements were difficult to understand and which will likely be the target of future critical change requests. The objective of this project is to provide multiple representations of the requirements and functionality of the system, which can be used to facilitate automated verification and validation of future changes and to facilitate re-engineering tasks.

Two major tasks were performed in the development of the formal specifications of the `Phase_Plane` high-level requirements. First, a concise description of the original requirements of the module was acquired. 'l'his information was obtained from the *Functional Subsystem Software Requirements* (FSSR) document [18] (also known as Level C requirements, consisting largely of "wiring" diagrams), the *Guidance and Control Systems Training Manual [19],* source code, informal design notes, and discussions with Shut tle software personnel. The resulting description was used to develop an "as-built" (implementation-biased) formal specification, capturing the functionality depicted in the FSSR wiring diagrams.

Second, in order to obtain a more abstract formal specification and eliminate the implementation bias present in the as-built layer, OMT diagrams were developed to represent the integral information from the low-level specifications. These diagrams facilitated the abstraction process and led to the higher-level specifications. This process of developing a level of formal specification, followed by the construction of the corresponding OMT diagrams, led to the identification of the high-level, critical requirements of the `Phase_Plane` module. Sample specifications and OMT diagrams arc described below.

3.1 Phase Plane

The *Reaction Control System* (RCS) Digital Auto Pilot system (DAP) achieves and maintains attitude through an error correction method, which involves jet firings. *Attitude* refers to the rotational position of the vehicle in terms of roll, pitch, and yaw. In order to make the Shuttle maintain a specific attitude, the crew specifies two values: attitude deadband and rate deadband. *Attitude deadband* refers to how

much drift (positive or negative) will be tolerated in any axis before jets are fired to correct the error. *Rate deadband* refers to the allowable rate changes of the attitude (positive or negative) before jet firings are required to null the error. Figure 2 gives a high-level view of the I)A}'. The *State Estimator* gives the current attitude, taking into consideration spacecraft dynamics. This information is supplied to the `Phase_Plane` component, which compares the attitude and rate errors (the rate of attitude change) with the desired (deadband) values specified by the crew.

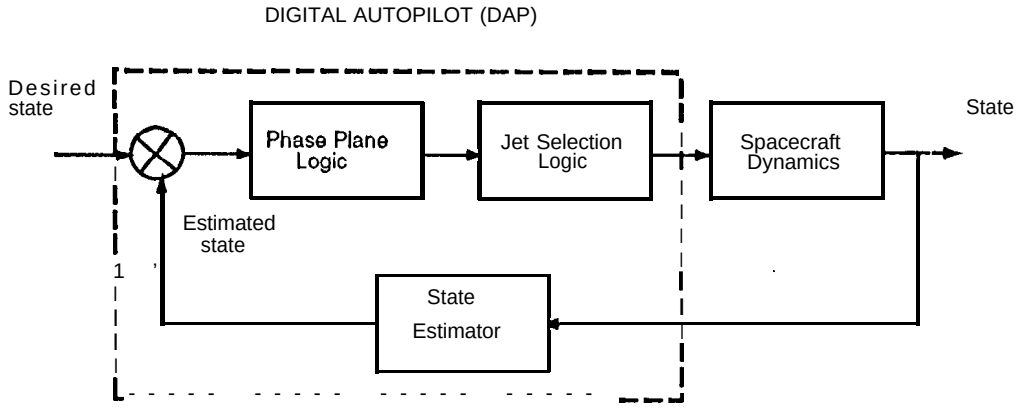

Figure 2: High-level view of I)Al', including the `Phase_Plane` module

Figure 3 gives a simplified graphical representation of the phase plane [1 8]. A phase plane is represented as a graph plotting spacecraft rate errors against attitude errors for one rotational axis, with a "box" (with parabolic sides) drawn around the center. A separate phase plane exists for each of the vehicle rotation axes (roll, pitch, and yaw). in an attitude hold situation, the error plot cycles around the zero error point with jets firing each time the limits of the "hex" are exceeded. This activity is known as "limit cycling" or "deadbanding". The `Phase_Plane` module generates positive or negative rate commands on an axis by axis basis, where the JetSelect component determines which jet(s) to fire (the topic of the larger multi-NASA site project [1 5]). The shaded *coast regions* depict, situations, where the Shuttle needs no corrective action. The remaining regions are known as *hysteresis regions*, where external factors, such as positive (negative) acceleration drift, propellant usage, inertia, time lags between firing commands, and sensor noise, are taken into consideration in order to preclude unnecessary jet firings, As such, the hysteresis regions are defined as a *function of* jet firings.

In Figure 3, the dashed lines outline the deadbanding path, whine cinch "()" indicates points that the Shuttle is changing state with respect to thruster firings; in this graph, the Shuttle transitions through

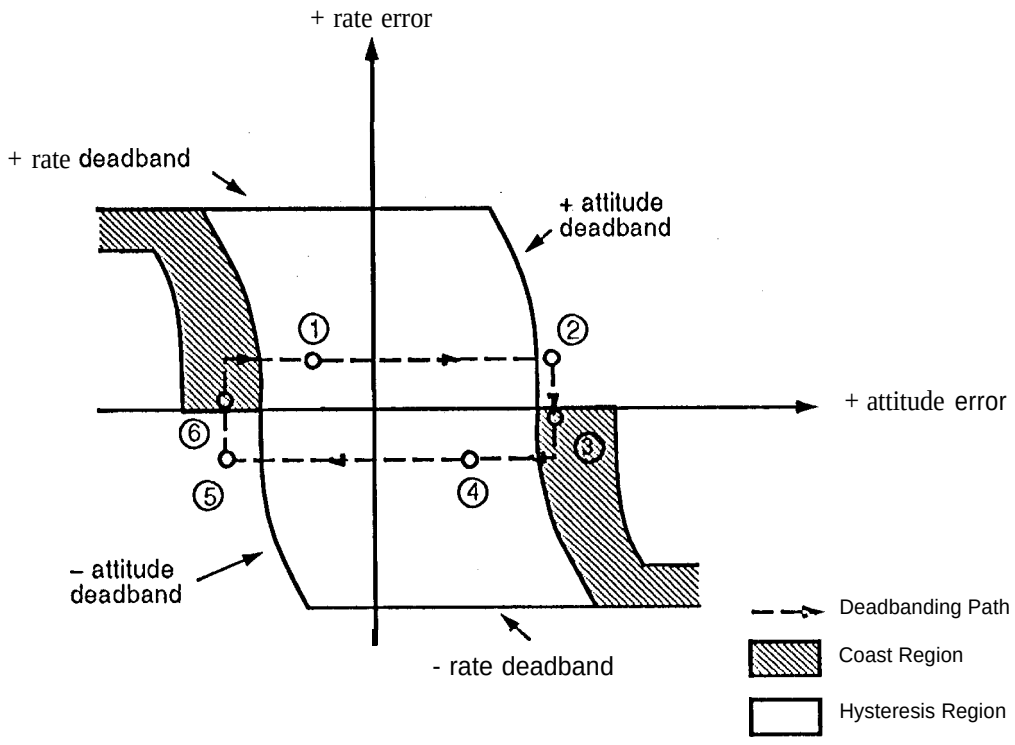

Figure 3: Graphical depiction of the phase plane, with coast and hysteresis regions [18]

six different states. As long as the current position is within the limits imposed by the deadbands, the deadband constraints are satisfied and no jets will be commanded to fire. Figure 4 gives an explanation of the different states in which the Shuttle can be while it is deadbanding [19].

The requirements for the `Phase_Plane` module are described in the FSSR document that includes a simplified wiring diagram (see Figure 5), which identifies the input and out put values, as well as several tables describing the calculation for the boundaries of the phase plane and its different regions. For historical reasons, the FSSR descriptions use notation commonly used for circuit design, even though the system being described is software-based. The solid lines represent data flows and dashed lines represent control. In Figure 5, the dashed line indicates that the *enable* flag must be set by the crew in order to enable the autopilot mode.

3.2 Formal Specifications

The software was formally specified using the *1'VS* (Prototype Verification Systems) terminal-based, formal specification tools [20] (e.g. syntax checker and theorem prover), which are under continuing development at SRI International. A PVS specification comprises a collection of *theories*. Each theory

1! No jets fire. Since the rate error is positive, the attitude error will grow in a positive direction,

2. Jets fire to nullify the positive rotational rate.

3. Jets stop firing when the deadband line is crossed, but a little negative rate errors is inevitable..

4. No jets fire. With a negative rate error, the attitude error will also drift negatively.

5. Jets fire to nullify negative rate error.

6. Jets stop firing, but residual positive rate error causes attitude error to go positive again and tile cycle repeats.

Figure 4: Explanation of deadbanding states [19]

consists of a *signature* for the type names and locally declared constants, as well as the axioms, definitions, and theorems associated with the signature,

in order to obtain a specification of the high-level requirements from the existing documentation and source code, several layers of specifications were constructed, where each layer is more abstract than the preceding layer. Specification of a system through increasingly detailed levels of abstraction is a well-established method [7, 21]. From the forward engineering perspective, the software development process proceeds in a top-down fashion. Typically, abstract, IIigh-level specifications arc used to establish the system inputs, outputs, and basic functionality; critical correctness requirements that the system must satisfy are stated at this level and become the criteria by which the specification is judged to be correct, Mid-level specifications introduce details of functionality and data structure requirements that may constrain the eventual implementation of the system; change requests for modules will most likely be addressed in these specifications. A 10W-1CVC1 specification is a straightforward representation of a particular implementation, which may be used to automatically generate source code [8, 17] or verification conditions for programmer-produced code [1].

In contrast, performing reverse engineering of the `Phase_Plane` project , involved a mixture of bottom-up with a top-down approach. This project explored the use of formal specifications to derive requirements that are more detailed and precise than an English paragraph, and less obscure than optimized source code. Specifications were developed in the following order: low-, high-, and mid-level, IIigh-level natural language descriptions of this portion of the Shuttle DAP were available, as was source code.

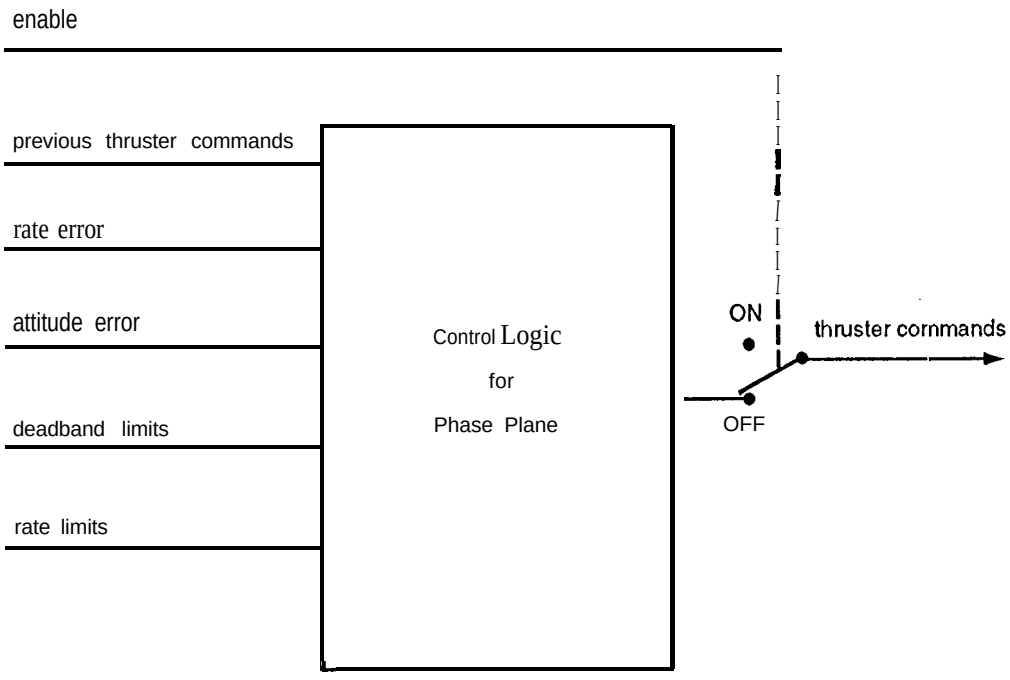

Figure 5: simplified wiring diagram for the Phase_Plane module [18]

Given the two types of documentation that varied in the amount of detail, we started with the low-level specifications to ensure that an accurate description of the current functionality was captured. Next, we used the high-level descriptions from the Crew Training Manual and constructed several OMT diagrams to assist in the specification of high-level requirements. Finally, in order to bridge the information gap bet wccn the low-level, implementation-specific and the high-level specific.ations, we constructed a set of mid-level specifications. We used the OMT diagrams to introduce abstraction into the low-level specifications, and we used he high-level specifications to identify critical properties applicable to the overall component in order to construct the mid-level specifications. The remainder of this section describes in more detail the specification process and includes example specifications.

The low-level formal specification of Phase-plane was developed from the existing source code, the Crew Training Manual [19], and the low-level wiring diagrams, This specification mirrored the functionality of the existing system, but did not offer an abstract view of the module's functional requirements. Due to space constraints, the low-level specifications are omitted but can be found in [22].

A high-level "black-box" specification was then developed, which did not include implementation details. At this level, it was straightforward to state abstract properties that any software implementing

Phase-Plane must possess. The high-level specification describes properties that characterize the Shuttle's position in terms of attitude and rate deadband values. If the Shuttle travels outside the specified regions, then the jets need to be fired to bring the Shuttle back into the phase plane region. Several data types were specified in the 10W-1CVCI specifications and are used for both the high- and mid-level specifications; they are given in Figure 6 for clarity purposes.

```
nonnegative-real:        TYPE = {x: real | x >= 0}
rate-error-type:         TYPE = real % unite are degrees/second
rate_deadband_type:      TYPE = non_negative_real
attitude_deadband_type:  TYPE = non-negative-real
attitude-error-type:     TYPE = real % units are degrees
```

Figure 6: Data types used for specifications of the Phase_Plane module

A few predicates are defined to describe general properties of the Shuttle, where Boolean predicates are denoted by a "?" suffix, and the types of the predicate arguments are enclosed in square brackets, First, the is_deadbanded? predicate determines whether the Shuttle is in a deadbanding state, where there are four arguments to the predicate corresponding to the attitude deadband, rate deadband, current attitude error, and current rate error represented by their respective types.

```
is_deadbanded? : pred[attitude_deadband_type ,rate_deadband_type,
                      attitude_error_type,rate_error_type]
```

Next, two predicates are defined to check whether rate and attitude errors are in a region where jets need to be fired to decrease rate error (generate positive! rate error).

```
decrease-rate-error? : pred[attitude_deadband_type,rate_deadband_type,
                            attitude_error_type ,rate_error_type]

increase-rate-error? : pred[attitude_deadband_type,rate_deadband_type,
                            attitude_deadband_type ,rate_deadband_type]
```

Figure 7 contains an abbreviated version of the top-level specifications. In this case, wiring_phase_plane refers to the low-level specifications. The referenced states are those depicted in Figure 3.

Based on the specification for the six states, the following high-level axiom was constructed to relate the attitude and rate deadbands, as well as the rate and attitude errors. Specifically, the axiom asserts that if the Shuttle is in the deadband regions, then there is no need to fire jets to increase or decrease

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Module:   High-Level Specifications of Properties for Phase Plane Module
%
% The following characterize the 6 states of Shuttle when it is deadbanding
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
high-level-phase-plane:  THEORY
BEGIN
  USING wiring_phase_plane % low-level specifications for phase plane
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% No jets fire. Since the rate error is positive, the attitude error will
% grow in a positive direction. (State 1)
%
no_jets_positive_rate?(att_db,rate_db,att_err,rate_err) : bool =
          is_deadbanded?(att_db,rate_db,att_err,rate_err) &
          rate-err > 0      &       att_err > 0
%
% Jets are firingto correct positive rotational rate (State 2)
%
jets_fire_correct_pos_attitude_error? (att_db,rate_db,att_err,rate_err):bool =
          NOT (is_deadbanded?(att_db,rate_db,att_err,rate_err) ) &
          decrease_rate_error?(att_db,rate_db, att_err,rate_err)
%
% Jets stop firing when deadband line is crossed, but a little negative
% rate error is inevitable. (State 3)
%
jets_stop_negative_rate_error?(att_db,rate_db,att_err,rate_err) : bool =
          is_deadbanded?(att_db,rate_db,att_err,rate_err) &
          rate-err < 0
%
% No jets fire. With negative rate error, the attitude error will also
% drift negatively. (State 4)
%
no_jets_negative_rate?(att_db,rate_db,att_err,rate_err) : bool =
          is_deadbanded?(att_db,rate_db,att_err,rate_err) &
          rate-err < 0 &       att_err < 0
%
% Jets are firing to correct negative attitude error (State 5)
%
jets_fire_correct_neg_attitude_error?(att_db,rate_db,att_err,rate_err) : bool =
          NOT (is_deadbanded?(att_db,rate_db,att_err,rate_err) ) &
          increase_rate_error? (att_db,rate_db, att_err,rate_err)
%
% Jets stop firing, but residual positive rate error causes attitude
% error to go positive again and cycle starts over (State 6)
%
jets_stop_positive_rate_error?(att_db,rate_db,att_err,rate_err) : bool =
          is_deadbanded?(att_db,rate_db,att_err,rate_err) &
          rate-err > 0

...

end  high-level-phase-plane
```

Figure 7: Sample high-level specifications of Phase-Plane

the rate error.

```
AXIOM FORALL
(att_db:attitude_deadband_type),(rate_db:rate_deadband_type),
(att_err:attitude_error_type), (rate_err:rate_error_type):
        is_deadbanded?(att_db,rate_db,att_err,rate_err) <=>
           NOT (decrease_rate_error?(att_db,rate_db,att_err,rate_err) OR
                increase_rate_error?(att_db,rate_db,att_err,rate_err)
               )
```

Finally, amid-level formal specification was outlined to capture critical aspects of functionality and requirements at a level that would be useful to Shuttle requirements analysts when reviewing proposed modifications to the module. Code developed from this specification would implement the "Phase Plane Logic" boxofthclow-level wiring diagram (Figure 5). The challenge at the mid-level was to omit extraneous implementation details, yet be precise enough to capture necessary properties concerning minimization of fuel usage, thruster firings, and movement about the desired attitude. In constructing tllcmid-level specifications, several assumptions were made. First, no external acceleration disturbances were taken into consideration. This assumption means that by taking advantage of symmetry, it is sufficient to specify only the upper (nonnegative rate error) half of the Phase_Plane diagram, as shown in Figure 8. Second, the hysteresis region was treated as a coast region. Finally, the specification dots not explicitly state that the software implementation is enabled by a flag set by the crew, nor does it state explicitly that the calculations will be done once for each axis (roll, pitch, and yaw),

In addition to those types already defined in the low-level specification (see Figure 6), new types were introduced in the mid-level specifications to represent absolute rate errors, thruster commands, and thruster acceleration types. Figure 9 gives the specification of new type declarations and external inputs.

A few utility functions are defined to simplify the specifications: absolute value, square, and sign.

```
abs(x: real):          real = IF x < 0 THEN -x ELSE x ENDIF
sqr(x: real):          non_negative_real = x*x
sign(x: real):         integer = IF x >= 0 then 1 else -1 ENDIF
```

Next, in Figure 10, we define a few deadbanding functions, where we take advantage of the symmetry and y represents the vertical axis (absolute value of rate error) and x is the horizontal (attitude error) axis. The symmetry property enables us to generalize the calculations to those in the upper half of the

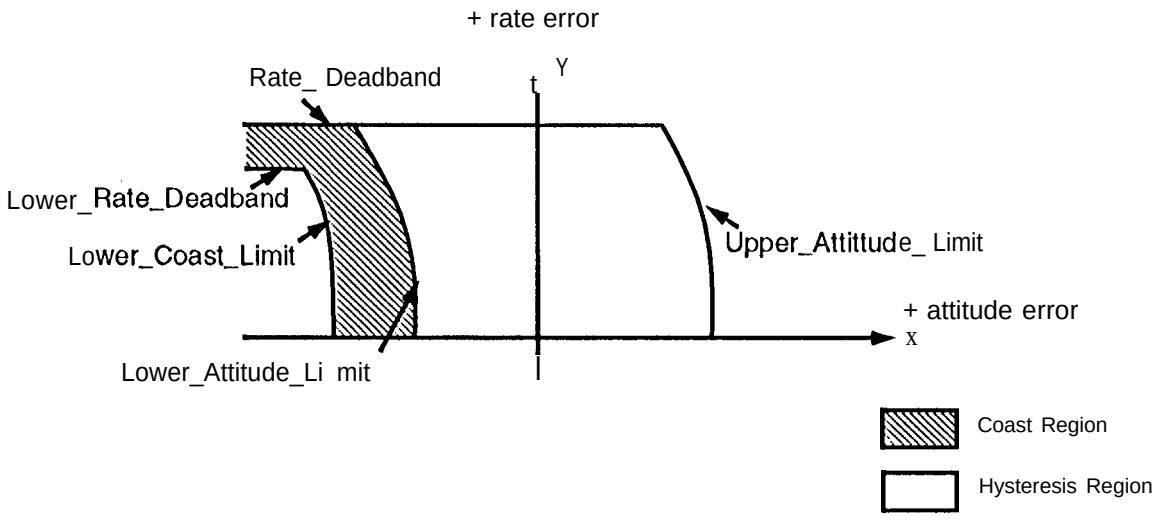

Figure 8: Upper Half of Phase Plane

deadband region. The adjust-for-symmetry function accounts for symmetry of the phase plane and returns the new thruster command given the current rate error and thruster command. The calculations for upper-attitude-limit and lower -attitude-1 imit are a generalization of a portion of the low-level specifications. These limits determine the bounds of the hysteresis regions, and, as mentioned previously, are a function of the jet firings.

The tail of the coast region is defined by the `rate_deadband` above and the `lower_rate_deadband` below. The `lower_rate_deadband` is typically O, `6*rate_deadband` [18]. The following specification gives the `lower_rate_deadband` as a real and asserts that the `lower_rate_deadband` is at most the `rate_deadband`.

```
lower. rate_deadband:        real
rate-de adband_relationship: AXIOM lower_rate_deadband <= rate_deadband
```

1

The lower (left) boundary of the coast region is defined by the `lower_attitude_limit` (a function declared below) and attitude -headband. The `lower_coast_limit` is typically lower-attitude-limit - 0. `2*a`
The specification asserts only that the lower-coast-limit is at most the `lower_attitude_limit`.

```
lower-coast-limit:           real
coast-limit-relationship:    AXIOM lower-coast-limit <= lower-attitude-limit
```

The primary function control-action returns a thruster command. Thruster hysteresis can be

```
%
% TYPE DECLARATIONS
%
   absolute_rate_error_type: TYPE = non-negative-real
   thruster-command-type:    TYPE = {positive-thrust, zero-thrust, negative-thrust)
   thruster_accel_type:      TYPE = non-negative-real
%
% External Inputs
%
% Rate and attitude deadbands characterize the desired bounds.
%
  rate_deadband:             rate_deadband_type
  attitude_deadband:         attitude_deadband_type
%
% Thrusters generate a constant acceleration during a firing period.
%
  thruster-impulse:          thruster_accel_type
%
% Rate and attitude errors are determined by on-board sensors.
%
  rate-error:                rate-error-type
  attitude-error:            attitude_error_type
```

Figure 9: Variable and tyl~e[leclaratiolls forlni[l-level specification

used to minimize thruster firings due to delays, sensor noise, or movement between state transition boundaries. At this level of abstraction, the hysteresis zone is treated the same as a "coast" zone. Figure 11 gives the specification for calculating the thruster commands. First, it must be determined if the spacecraft is outside the deadband area and thrusters should be fired "downward". Second, it must be determined whether the spacecraft is outside the deadband area and thrusters should be fired "upward'' 'J'bird, if the spacecraft is within the "coast" zone, then do not fire thrusters. If all the above cases do not apply, then incorporate thruster hysteresis.

3.3 Construction of OMT Diagrams

Since the original Phase-plane software was not object-oriented, the OMT analysis began with the source code and implementation-specific wiring diagram of the `Phase_Plane` module and resulted in two levels of data flow diagrams. These dia.grains assisted in the abstraction process to obtain an architectural view of the phase plane as it related to the overall DAP system, thus leading to the construction of the object models. Using the functional and object diagrams in conjunction with the description of the deadbanding states, we created the dynamic model for the `Phase_Plane` module. The dynamic model depicts the states between jet firings as the Shuttle deadbands. A high-level of

```
%
% Calculate coordinates for plotting attitude and rate errors
%
  y: absolute-rate-error-type = abs(rate_error)
  x: real = sign(rate_error)*attitude_error

% Because all calculations are done in the upper half of the deadband
% region, the calculated thruster command may need to be reversed.

  adjust_for_symmetry(t: thruster_command_type,
                      re: rate-error-type) : thruster-command-type =
      IF (t = zero-thrust) OR (sign(re) >= 0)
         THEN t
         % re was negative, so thruster commands must be reversed
         ELSE IF t = positive-thrust
                 THEN negative-thrust
                 % t was negative-thrust
                 ELSE positive-thrust
              ENDIF
      ENDIF
%
% Calculate boundary of hysteresis region based on a functionof jet firings
%
  upper-attitude-limit: real = -sqr(y)/(2*thruster_impulse) + attitude_deadband
  lower-attitude-limit: real = -sqr(y)/(2*thruster_impulse) - attitude_deadband
```

Figure 10: Variables and deadbanding functions to adjust for symmetry in phase lane

specifications was generated based on the dynamic model. The object and the functional models offered one level of abstraction, thus leading to the development of the next layer of formal specifications (mid-level specifications describing data structures and operations on the data structures).

The remainder of this section describes the OMT diagrams constructed during the reverse engineering and forma] specification process.

3.3.1 Functional Models

Data flow diagrams (DFD) facilitate a high level understanding of systems and are used in both forward and reverse engineering. Static analysis of program code provides information that accurately describes flow of data in a system. Process "bubbles" denote procedures or functions sofa given system, arrows represent data flowing from one process to another, and rectangles represent external entities.

The simplest functional model is a *context diagram*, or Level O DFD; the Level O DFD for the Phase-plane module is shown in Figure 12, where the entire phase plane module is reduced to a process bubble, with the external input and output labeled. The Level O DFD closely resembles the structure of the wiring diagram for Phase_Plane given in Figure 5.

```
thruster-hysteresis:       thruster-command-type = zero-thrust

control-action: thruster-command-type =
  IF (y > rate_deadband) OR (x > upper_attitude_limit)
     THEN adjust_for_symmetry(negative_thrust,rate_error)
     ELSE IF (y <lower_rate_deadband) ANO (x <lower_coast_limit)
             THEN adjust_for_symmetry(positive_thrust, rate-error)
             ELSE IF      (y <= rate_deadband)
                      AND (lower_rate_deadband <= y)
                      AND (x <= lower_attitude_limit)
                  OR      (x <= lower-attitude-limit)
                      AND (lower-coast-limit <= x)
                      ANO (lower_rate_deadband <= y)
                      THEN zero-thrust
                      ELSE thruster-hysteresis
                  ENDIF
          ENDIF
  ENDIF
```

Figure 11: Specification of Function to Calculate Thrust Commands

Figure 13 gives the next level DFD, which shows the different processes that constitute the Phase_Plane module. As shown in this figure, the input variables are used to calculate boundaries for the phase plane. The boundaries, the attitude deadband and the rate deadband, are supplied to the Phase_Plane module, which calculates thruster commands (jet firings). The thruster commands are then supplied to the Jet_Select module that determines which combination of jets should be used to achieve the desired thruster effect.

3,3.2 Object Models

Development of the as-built layer of specifications, the DFDs, and the requirements document for Phase_Plane led to the development of an object model for the Phase_Plane.

Figure 14 depicts a high-level object model for the entire DAP, consisting of the State Estimator, Phase Plane, and the Jet Select classes, corresponding to the diagram given in Figure 2. Each class consists of three parts corresponding to the name of the class, list of attributes, and list of operations, respectively. The diamond symbol demotes aggregation, where the class above the diamond is said to consist of the three classes below the diamond. If either attributes or operations are not known (or do not exist) for a given class, then the corresponding area is shaded. The Phase Plane class uses the class Crew Supplied Information, which represents the deadband limits that are used in the calculation of the

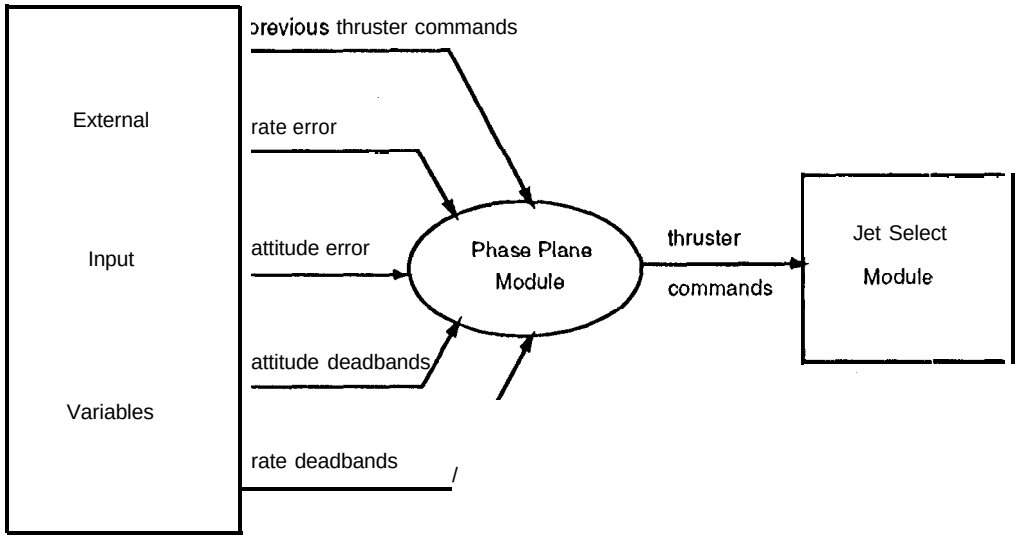

Figure 12: High Level (0) DFD for Phase_Plane Module

phase plane boundaries.

Figure 14 also contains the object diagram for the Phase Plane class, with attributes *rate error*, *attitude error*, and *rotation axis*. The operation for this class is *calculate thrust commands*, based on the difference between the current rate and attitude error values and those respective limits supplied by the crew. The filled circle attached to the Phase Plane class, indicates that the DAP contains three phase plane components, one to calculate different thrust commands for each of the specific rotational axes: roll, pitch, and yaw. There are two components for each Phase Plane class, Coast Region and Hysteresis Region. In the coast region, only the values of the attitude and rate errors are used to determine whether the Shuttle is still within the deadband limits. In the hysteresis region, however, additional information, such as fuel usage, sensor noise, and other spacecraft dynamics, is used to calculate thrust commands,

3.3.3 Dynamic Models

For completeness sake with respect to three models of OMT, this section gives the dynamic models for the phase plane, which describes the states in which the DAP can be with respect to the Phase-Plane component. Also included are the transitions that take the DAP from one state to another. A pictorial diagram of the position of the Shuttle is given in Figure 3. Since the Phase_Plane module is an event-based system, the state transition diagram is straightforward to construct.

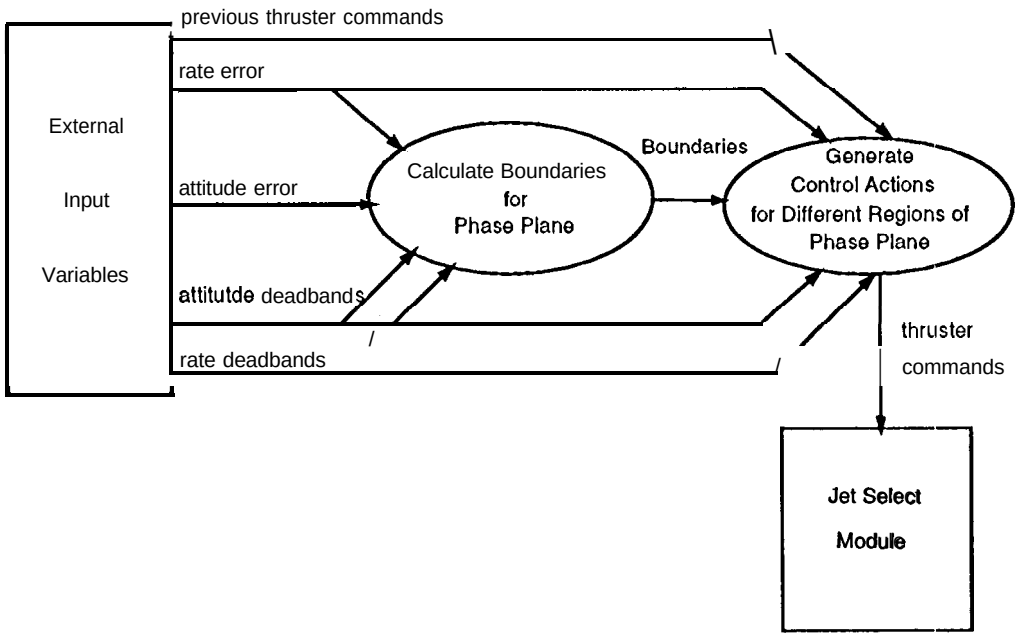

Figure 13: Level 1 DFD for Phase-plane Module

Figure 15 gives a statechart depiction of the states through which the Shuttle transitions while it is deadbanding. The state transitions are in the form of jets terminate (begin) firing and the Shuttle drifting in (out) of the deadband region.

Note that Figure 3 depicts the clockwise traversal of the states in which the Shuttle cycles through the deadband limits. It is also possible for the Shuttle to traverse the cycle in a counterclockwise fashion, in which case, the arrows in Figure 15 would be reversed.

Finally, a very high-level view of the states in which the Shuttle can be is given in Figure 16. Included in the diagram are the actions or conditions that cause the Shuttle to transition from one state to the next: jet firings and drift. The rectangle containing "Phase Plane" and the labeled arrows pointing to the states indicate that the state transitions describe the `Phase_Plane` module.

4 Lessons Learned

The results from this reverse engineering project have provided several lessons for the overall Space Shuttle project as well as for future reverse engineering projects. First, in order to obtain high-level requirements for existing software, it is difficult to obtain the specifications (formal or informal) in a single step. Instead, several layers of specifications should be developed, starting with the as-built

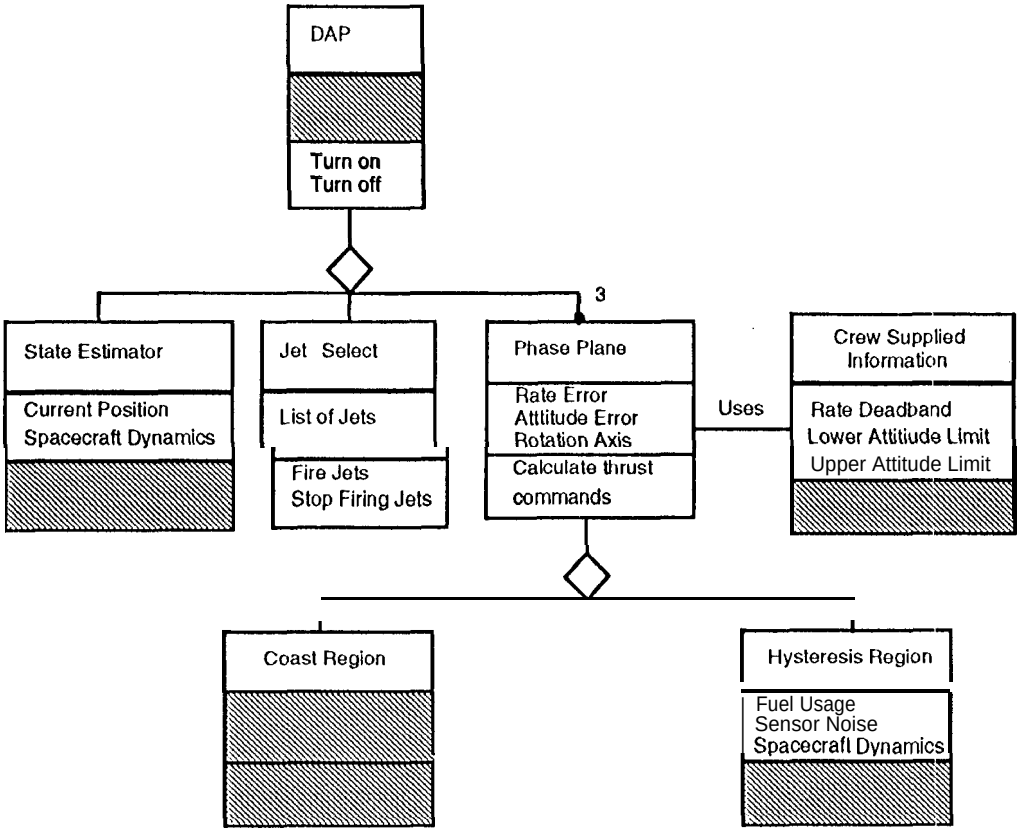

Figure 14: Object model for DA]'

specification, By closely mirroring the programming structure of the existing software, this specification provides traceability through the different levels of specifications.

Second, formal specification languages and their corresponding reasoning systems provide a framework for integrating disparate sources of project information to describe a system at many levels of detail. The project information may be documented in a variety of formats, from different sources, and subjected to varying levels of forma] review. For this particular project, information was obtained from implementation-specific wiring diagrams, definitions and instructions from a crew training manual, source code, informal design notes, and discussions with shuttle software personnel. The information was analyzed and distilled into specifications and OMT diagrams. The *PVS* proof system provided a mechanism for checking the completeness and consistency of the specifications.

'1'bird, the benefits of ob jcct-oriented analysis can be exploited for reverse-engineering as well as forward engineering projects. Specifically, object-oricllte.d analysis assists in the understanding of large, complex systems. Furthermore, an object-oricmtcd perspective facilitates future modifications

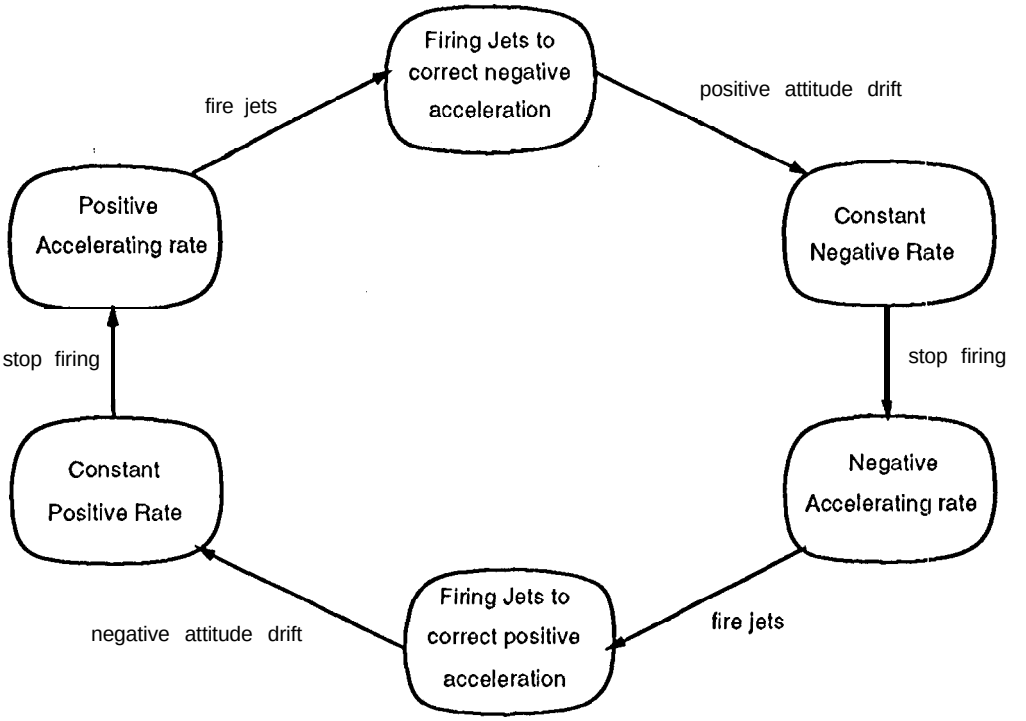

Figure 15: States representing the clockwise deadbanding of the Shuttle

by providing the requirements analyst and the developer with a high-level, abstract view of system components.

Finally, a process consisting of the construction of a level of formal specifications, followed by a set of corresponding diagrams, is needed to develop several layers of specifications for an existing system. The diagrams introduce abstractions that can be used to guide the construction of the next level of specifications. Furthermore, the three complementary notations in the OMT approach enable the

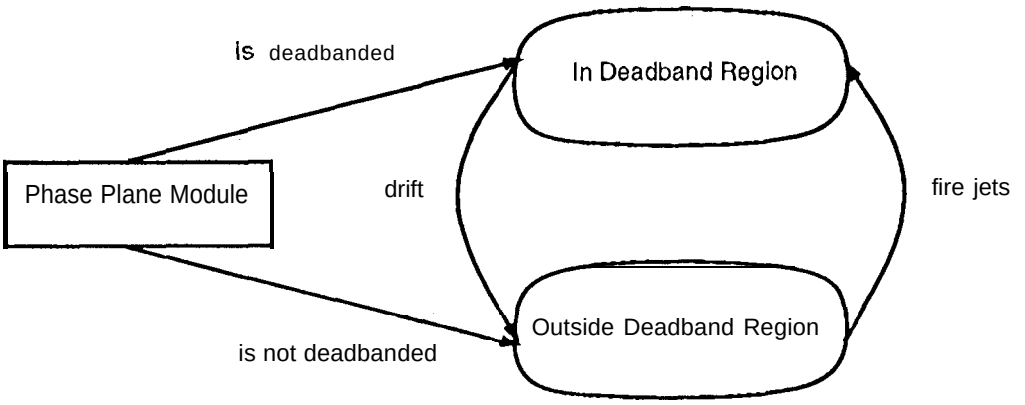

Figure 16: High-level states for Orbiter with respect to the Phase_Plane module

specifier to represent different components of the system using the Imst-suited type of diagram,

5 Conclusions and Future Investigations

Using formal specifications and object-oriented analysis to describe the software that implements the Phase-Plane module of the Space Shuttle DAJ' has demonstrated that these complementary analysis and development techniques can be used for existing, industrial applications. Constructing the different levels of specifications, with increasing abstraction, supplemented by the OMT diagrams provided a means for integrating different types of information regarding the `Phase_Plane` module from disparate sources. Having access to the formal specifications and diagrams will facilitate the verification that the original (critical) requirements or properties are not violated by any future changes to the software. in addition to facilitating verification tasks, the formal specifications can be used as the basis for any automated processing of the requirements, including checks for consisten cy and completeness, interaction with the requirements analyst and other members of the original development team for the project strongly support the conclusion that the specification construction process is useful to the overall software development and m aintenance processes of legacy (safet y-critical) systems [15].

Future investigations will continue to refine the mid-level and high-level specifications and develop theorems to relate the levels of specifications, We are also investigating the formalization of the OMT diagraming notation, which will provide a means for using automated techniques for extracting formal specifications from the OMT diagrams in order to fiat.ilitate the specification process [23]. Furthermore, extracting the specifications directly from the diagrams will enable us to reason about the completeness and consistency of the diagrammed system, thus facilitating the requirements analysis, design, and maint enance phases of software development.

6 Acknowledgements

Several people have provided valuable information and assistance during the course of the project. Specifically, we would like to thank Rick Covington, David Hamilton, John Kelly, Philip McKinley, and John Rushby.

Reference herein to any specific commercial product, process, or service by trade, name, trademark,

manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California, Institute of Technology.

References

[1] J. Rushby, "Formal methods and the certification of critical systems," Technical Report SIU-CSI,-93-07, SRI International, Computer Science laboratory, 333 Ravenswood Ave., Menlo Park, CA 94025-3493, November 1993. Available via anonymous ftp from f tp, cs1. sri. corn.

[2] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 Accidents," *IEEE Computer*, pp. 18-41, July 1993.

[3] P. G. Neumann and contributors, "Risks to the public,>' in *Software Engineering Notes*, ACM Special Interest Croup on Software Engineering,] 993.

[4] Aeronautic.s and Space Engineering Board National Research Council, *An Assessment of Space Shuttle Flight Software Development Practices. National* A cadem y Press, 1993.

[5] R. A. Kemmerer, "Integrating Formal Methods into the 1 Development Process," *IEEE Software*, *pp.* 37-50, September 1990.

[6] S. Gerhart, D. Craigen, and T. Ralston, "Experience with Formal Methods in Critical Systems and Regulatory Case St u dies," *IEEE Software, vol. 11,* January 1994.

[7] J. M. Wing, "A Specifier's introduction to Formal Methods," *IEEE Computer, vol. 23, pp. 8-24,* September 1990.

[8] 11.11. C. Cheng, "Synthesis of Procedural Abstractions from Formal Specifications," in *Proc. of COMPSAC'91*, pp. *149-154,* September 1991.

[9] E. J. Chikofsky and J. 11. Cross 11, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, January 1990.

[10] E. J. Byrne and D. A. Gustafson, "A Software Re-engineering Process Model ," in Proceedings *of COMPSAC'92: Computer Software and Applications Conference*, *pp.* 25-30, September 1992.

[11] B. H. C. Cheng and G. C. Gannod, "Constructing formal specifications from program code," in *Proc. of Y'bird International Conference on Tools in A rtificial Intelligence*, *pp. 125-128,* November 1991.

[12] G. C. Gannod and 11.110 C. Cheng, "A two-phase approach to reverse engineering using formal methods," *Lecture Notes in Computer Science, Proc. of Formal Methods in* Programming *and Their Applications Conference,* vol. 735, pp. 335-348, June 1993.

[13] G. C. Gannod and B. 11. C. Cheng, "Facilitating the maintenance of safety-critical systems," *Int. J. of Software Engineering and Knowledge Engineering,* vol. 4, no. 2, pp. 183-204, 1994.

[14] J. C. Kelly, R. G. Covington, and D. Hamilton, "Results of a forma] methods demonstration project ," in *Proc. of WESCON (to appear)*, (Anaheim, California), September 1994.

[15] Jet Propulsion Laboratory, Johnson Space Center, and Langley Research Center, "Formal Methods Demonstration Project for Space Applications:]'base I Case Study: STS Orbit DAP Jet Select." December 1993.

[16] J, Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W, Lorensen, *Object-Oriented Modeling and Design,* Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[17] B. 11. C. Cheng, "Applying formal methods in automated software development," *Journal of Computer and Software Engineering,* vol. 2, no. 2, pp. 137-164, 1994.

[18] "Space Shuttle Orbiter Operational Level C Functional Subsystem Software Requirements: Guidance Navigation and Control -- Part C Flight Control Orbit DAP," Tech. Rep. 01-21 edition, Rock well international, Space Systems Division, February 1991.

[19] S. Beck, "G & C Systems Training Manual: Guidance and Flight Control - Insertion, Onorbit and Deorbit," Tech. Rep. I/O/D G&C21 02, Mission Operations Directorate, Training Division, Flight Training Branch, October 1985.

[20] N. Shankar, S. Owre, and J. Rushby, "The 1'VS specific.ation language and tools," technical report, Computer Science laboratory, 333 Ravenswood Ave., Menlo Park, CA 94025 -3493, 1993. Available via anonymous ftp from ftp, `csl.` sri, corn.

[21] C. B. Jones, *Systematic Software Development Using VDM.* Prentice Hall international Series in Computer Science, Prentice Hall International (UK) Ltd., second ed., 1990.

[22] B. H. C. Cheng and B. Auernheimer, "Applying Formal Methods and Object-Oriented Analysis to the Existing Space Shuttle Software," Technical Report CPS-94-9, Michigan State University, Department of Computer Science, A714 Wells Hall, East Lansing, Michigan 48824, February 1994.

[23] R. 11. Bourdeau and 11.11, C. Cheng, "A formal semantics of object models," Technical Report MSU-CPS-94-6, Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, 48824, January 1994. (submitted to *IEEE Trans. on Software Engineering.).*